On Adopting Linters to Deal with Performance Concerns in Android Apps

Sarra Habchi, Xavier Blanc, Romain Rouvoy

On Adopting Linters to Deal with Performance Concerns in Android Apps

Sarra Habchi
Inria
University of Lille
Lille, France
sarra.habchi@inria.fr

Xavier Blanc
University of Bordeaux
Bordeaux, France
xavier.blanc@u-bordeaux.fr

Romain Rouvoy
University of Lille
Inria
Lille, France
romain.rouvoy@inria.fr

ABSTRACT

With millions of applications (apps) distributed through mobile markets, engaging and retaining end-users challenge Android developers to deliver a nearly perfect user experience. As mobile apps run in resource-limited devices, performance is a critical criterion for the quality of experience. Therefore, developers are expected to pay much attention to limit performance bad practices. On the one hand, many studies already identified such performance bad practices and showed that they can heavily impact app performance. Hence, many static analysers, *a.k.a.* linters, have been proposed to detect and fix these bad practices. On the other hand, other studies have shown that Android developers tend to deal with performance reactively and they rarely build on linters to detect and fix performance bad practices. In this paper, we therefore perform a qualitative study to investigate this gap between research and development community. In particular, we performed interviews with 14 experienced Android developers to identify the perceived benefits and constraints of using linters to identify performance bad practices in Android apps. Our observations can have a direct impact on developers and the research community. Specifically, we describe why and how developers leverage static source code analysers to improve the performance of their apps. On top of that, we bring to light important challenges faced by developers when it comes to adopting static analysis for performance purposes.

CCS CONCEPTS

• **Software and its engineering → Software performance**; *Software maintenance tools*;

KEYWORDS

Android, performance, linters, static analysis.

1 INTRODUCTION

Mobile applications (apps) are nowadays complex software systems that must be designed carefully to meet the user expectations and stay ahead of the app stores competition. Reports show that 75 % of apps are uninstalled within 3 months [2] and that the second top reason of these uninstalls is poor performance [1]. Therefore, app developers are expected to pay a careful attention to performance in their development, and particularly avoid performance bad practices. Previous studies have already identified and characterised different development practices that hinder the performance of mobile apps [20, 21, 29, 35]. Researchers also assessed the impact of such bad practices on different performance aspects [16, 22]. To detect these bad practices, different static analysers, *a.k.a.* linters, like Paprika [23], PerfChecker [29], and aDoctor [32] were proposed. The development community also proposed tools to detect and fix these bad practices. For instance, Android Studio, the official IDE for Android development, integrates a linter—called Android Lint—that detects performance bad practices.

However, despite the availability of these linters and evidences of performance penalties due to bad practices, Android developers do not rely heavily on linters to deal with performance concerns. A survey conducted by Linarez *et al.* [27] with 485 Android developers showed that, when using tools to manage performance, most developers rely on profilers and framework tools, and only 5 participants reported using a linter. In order to confirm this phenomenon and lay the foundation for our study, we published an online survey and asked Android developers about their linter usage. All the details of the survey and its results are available in our technical report [12]. The results of this survey reported that only 51 % of Android Lint users rely on it for performance purposes. Given that performance checks are enabled by default in Android Lint, such observations raise many questions about how Android developers perceive the usefulness of linters when it comes to performance. In particular, it is important to highlight the benefits and challenges of adopting linters for performance issues.

In this paper, we therefore conduct a qualitative study to investigate the benefits and constraints of using linters for performance purposes. We interview 14 experienced Android developers who use Android Lint for performance purposes in order to understand:

(1) Why do Android developers use linters for performance purposes?

(2) What are the constraints of using linters for performance purposes?

It is also important to understand what fashions would allow Android developers to achieve the eventual benefits of linters for performance, thus we also investigate the question:

(3) How do Android developers use linters for performance purposes?

Our findings from this study have direct implications to developers, researchers and tool makers.

The remainder of this paper is organised as follows. We start with a brief background about linters for Android in Section 2, then we describe our methodology in Section 3. Section 4 reports and discusses the results of our qualitative study. We identify in Section 5 the implications of our results, and in Section 6 the limitations of our study. We present in Section 7 the related works before concluding in Section 8.

2 BACKGROUND

This section provides general background information on linters for Android. Many linters, like PMD [11], CheckStyle [5], Infer [9], and FindBugs [8] can be used to analyse Android projects written in Java. PMD detects programming flaws like dead code. CheckStyle checks that coding conventions are respected. Infer identifies issues as potential bugs, null pointer exceptions, resource leaks, etc. FindBugs analyses the Java bytecode to detect potential bugs. Detekt [6] is a linter that can be used on Android projects written in Kotlin. It computes source code complexity and identifies some code smells. Ktlint [10] is another linter for Kotlin, but that focuses on checking code conventions. All the mentioned linters detect issues related to either Java or Kotlin, but they do not consider issues or practices specific to the Android framework.

Android Lint [4] is the mainstream linter for Android. It is integrated in Android Studio, the official IDE for Android. It can be run on Android projects from the command line or in Android Studio interactively. It scans the code to identify structural code problems that can affect the quality and performance of Android apps.

Lint targets 339 issues related to correctness, security, performance, usability, accessibility, and internationalization. The category performance includes 34 checks,[1] *a.k.a.* rules. As an example, we explain the rule HandlerLeak that checks if a `Handler` is used as non-static inner class or not. This situation is problematic because the `Handler` holds a reference to the outer class. Thus, as long as the `Handler` is alive the outer class cannot be garbage collected, thus causing memory leaks. This issue has been addressed in some research studies as a code smell, named *Leaking Inner Class* [23]. Android Lint reports each problem with a brief description message, a priority, and a severity level. The priority is a number from 1 to 10, and the severity has three levels, `ignore`, `warning`, and `error`. All the Android Lint checks have a default priority and severity. The severity is a factor that can be configured by developers to classify the problems on which they want to focus.

We chose to use Android Lint in this study as it is the most used linter for Android, and it detects a large set of performance bad practices.

3 METHODOLOGY

Our objective is to investigate with an open mind the benefits and limitations of using a linter for performance purposes in Android. Therefore, we follow a qualitative research approach [18] based on classic Grounded Theory concepts [13]. With this approach, we aim to discover new ideas from data instead of testing pre-designed research questions. Specifically, we conducted interviews with 14 experienced Android developers. The interview design and the selected participants are presented in Sections 3.1 and 3.2, respectively. Afterwards, we transcribed and analysed the interviews, as explained in Section 3.3.

3.1 Interviews

As commonly done in empirical software engineering research, we designed semi-structured interviews. This kind of interviews consists of a list of starter questions with potential follow up questions that are asked through out. We followed the advices given by Hove and Anda *et al.* [24] to design and conduct the interviews. In particular, we paid a careful attention to explaining the objectives of the interviews. We explained that interviews are not judgemental and we incited the participants to talk freely. We asked open-questions, such as: *"Why do you think that Lint is useful for performance purposes?"*, and we asked for details whenever possible. We designed the interview with basically 12 questions, and depending on participants' replies, we asked additional questions to explore interesting topics. The main questions are the following:

(1) Why do you use the linter to deal with performance concerns?
(2) What are the benefits that you perceived with this usage?
(3) How do you use the linter to deal with performance concerns?
(4) How do you configure the linter?
(5) Do you use it individually or in a team?
(6) In a collaborative project, how do you configure the linter to optimize the app performance?
(7) Do you integrate the linter in the build or CI? Why?
(8) Do you change the priority or severity of performance checks? Why?
(9) Do you ignore or suppress performance checks sometimes? Why?
(10) Are there any performance checks that you consider irrelevant? Why?
(11) Do you write your own performance checks? Why?
(12) In your opinion, what are the constraints of using the linter for performance purposes?

With the permission of interviewees, the interviews were recorded and they lasted from 18 to 47 minutes, with an average duration of 30 minutes. We performed two interviews face-to-face, and the 12 others were performed with an online call. One participant was not able to participate in the call and instead received a list of questions via email and gave written answers.

3.2 Participants

Our objective was to select experienced Android developers who use the linter for performance. We did not want to exclude *open-source software* (OSS) developers, nor developers working on commercial projects. For that purpose, we relied on many channels to contact potential participants.

GitHub. First, we selected the most popular Android apps on GitHub relying on the number of stars. Afterwards, we selected the projects that use the linter by looking for configuration files —*e.g.*, `lint.xml`, or configurations in Gradle files. Then, we manually analysed the top-100 projects to identify the developers who actively work with the linter. We only found 41 developers that contributed to the linter configuration. As it is complex to guess if developers are motivated to use the linter for performance only from the configuration files, we contacted these developers to ask them if they use the linter for performance or not. Out of 41 mails sent, we received 18 answers, *i.e.*, a response rate of 43 %. 13 developers answered that they use the linter for performance, and 5 others answered negatively. We replied to the 13 developers to explain the objectives of our interview and invite them to participate. We

[1] As for March 2018.

received 6 acceptances and 2 rejects, the other developers did not answer.

Forums and meetups. To select commercial Android developers, we sent forms in developer forums [3]. In the forms, we explicitly explained that we are looking for Android developers who use the linter for performance. We received 6 answers from forums, 1 of them was irrelevant because the developer did not have a real experience with Android Lint. We also communicated the same message in Android development meetups. From meetups, we selected 3 persons who satisfied our criteria.

Overall, this selection process resulted in 14 participants. After conducting 14 interviews, we considered that the collected information is enough to provide us with theoretical saturation [19], *i.e.*, all concepts in the theory are well-developed. Thus we did not perform a second batch of selection and interviews.

To keep the anonymity of the 14 selected participants, we refer to them with code names. We also omit all the personal information like company or project names. Table 1 shows the participants codes, their experience in Android and Android Lint in terms of years of professional development, and the types of projects they work on. It is worth mentioning that with the term "commercial" we refer to projects that are developed in an industrial environment and are not based on an open-source community. Also we found that all the developers selected from GitHub were also involved in commercial projects, and two developers spotted from forums were involved in both commercial and OSS projects

Table 1: Partcipants' code name, years of experience in Android and Android Lint, and the type of projects they work on

Participant code	Android experience	Lint experience	Project type
P1	8	5	OSS & Commercial
P2	8	4	OSS & Commercial
P3	8	4	Commercial
P4	8	5	OSS & Commercial
P5	8	4	OSS & Commercial
P6	8	5	OSS & Commercial
P7	6	4	OSS & Commercial
P8	5	3	OSS & Commercial
P9	4	4	Commercial
P10	5	3	Commercial
P11	2	2	Commercial
P12	5	4	Commercial
P13	4	1	Commercial
P14	8	4	OSS & Commercial

Table 1 shows that, out of 14 participants, 11 have more than 5 years of experience in Android. Compared to the age of the Android framework (8 years), this experience is quite strong. As for Android Lint, which has been introduced with Android Studio in 2013, 10 of our participants have more than four years of experience in using it.

3.3 Analysis

We carefully followed the analytical strategy presented by *Schmidt et al.* [37], which is well adapted for semi-structured interviews. This strategy has proved itself in the context of research approaches that postulate an open kind of theoretical prior understanding, but do not reject explicit pre-assumptions [37]. Before proceeding to the analysis, we transcribed the interviews recordings into texts using a denaturalism approach. The denaturalism approach allows us to focus on informational content, while still working for a *"full and faithful transcription"* [31]. In what follows, we show how we adopted the analytical strategy steps.

3.3.1 Form material-oriented analytical categories. In this step, we define the semantic categories that interest us. In our case, we investigate the motivation and arguments of Android developers that use the linter for performance purposes, and the constraints of such usage. Therefore, our categories are initially the following two topics:

- Why do Android developers use linters for performance purposes?
- What are the constraints of using linters for performance purposes?

After our discussions with participants, we noticed that an additional category is highlighted by developers, namely:

- How do Android developers use linters for performance purposes?

We found this additional topic enlightening, thus we included it in our analytical categories. Once the categories are set, we read and analysed each interview to determine which categories it includes. In the analysis, we do not only consider answers to our questions, but also how developers use the terms and which aspects they supplement or omit. After this analysis, we can supplement or correct our analytical categories again.

3.3.2 Assemble the analytical categories into a guide for coding. We assemble the categories into an analytical guide, and for each category different versions are formulated. The versions stand for different subcategories identified in the interviews in reference to one of the categories. In the following steps, the analytical guide can be tested and validated. Indeed, the categories and their versions may be refined, made more distinctive or completely omitted from the coding guide.

3.3.3 Code the material. At this stage, we read the interviews and try to relate each passage to a category and a variant formulation. As we focus on labelling the passages, we may omit special features and individual details of each interview. Yet, these details will be analysed and highlighted in the last step. To strengthen the reliability of our conclusions, we use a consensual coding. Therefore, each interview is coded by at least two authors. Initially every author codes the interview independently, afterwards, the authors discuss and compare their classification. In case of discrepancies, the authors attempt to negotiate a consensual solution.

4 RESULTS

We present in Table 2 the results of the coding process in order to contribute to the transparency and verifiability of our study.

Table 2: Results of the coding process.

Categories	Versions (subcategories)	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
Benefits	Learn about the framework	x	x	x				x		x	x	x			
	Anticipate performance bottlenecks	x	x			x	x				x			x	
	The linter is easy to use		x		x		x		x			x			x
	Develop the performance culture	x				x		x			x				
	Save time		x	x						x					
	Contribute to credibility and reputation	x					x	x							
	Raise awareness of app performance			x		x									
	Ease performance debugging									x					
Fashions	Integrate from scratch		x			x				x	x	x		x	
	Target milstones	x				x	x	x							
	Adopt performance sprints					x							x		
	Improve performance incrementally									x					
	Support individual development	x		x		x			x	x		x		x	
	Check performance rules in a team		x		x		x	x		x	x		x		x
	Prioritise performance aspects		x	x		x	x			x				x	
Constraints	Static analysis is not suitable for performance	x	x	x	x			x					x		
	Nobody complained	x		x	x	x			x	x			x		
	We do not have time	x					x	x							
	Performance is not important in our case	x							x				x		
	Some rules are irrelevant		x										x		
	Results are not well presented	x			x		x					x		x	x
	The linter lacks precision									x	x			x	
	The linter does not have enough checks					x	x			x					
	It is difficult to write new rules		x												

4.1 Why do Android developers use linters for performance purposes?

4.1.1 **Linter helps developers to learn about the framework.** As the Android framework is relatively young, developers are eager to learn about its underlying performance constraints. Half of the participants stated that the linter is very instructive in that respect (P1, P2, P3, P7, P9, P10, P11). *"Lint will actually help you become a better Android programmer"* (P3). Indeed, the performance checks of the linter mentor the developers to use the framework efficiently: *"I see the performance checks as a guide of the Android framework"* (P7). Other participants mentioned that their understanding of performance, and their programming habits evolved thanks to the linter: *"Everytime I learn a new thing about performance, and then it becomes a habit"* (P11). As an example of these cases, participant P9 mentioned DrawAllocation, a bad practice that consists of allocating new memory spaces in the `onDraw()` method: *"I was creating a new object in a paint method so Lint gave me a warning. That was the first and last time I see it, because now I pay attention to this"* (P9).

Some participants emphasised that junior Android developers should particularly use the linter for performance: *"Lint checks are extremely useful for helping out beginner Android developers or junior members of my team to enforce better code performance"* (P2). Indeed, junior Android developers, even if they have a prior experience in desktop development, may lack understanding of mobile framework specificities. Thus, they are prone to performance-related bad practices and they need the linter to learn how to keep their mobile apps efficient.

Discussion. Participants from previous studies have also reported that learning is one of the main benefits of using linters [17, 38]. Specifically, the developers learned with the linter about the syntax of the programming language, idioms, or libraries. In the case of our study, the participants showed that even important concepts about the Android framework can be learned through the linter. This can be a great incentive for Android developers to use linters with performance and other framework related checks.

4.1.2 **Linter can anticipate some performance bottlenecks.** Many participants reported that the linter supports them in detecting bad practices that can cause performance bottlenecks (P1, P2, P5, P6, P10, P13). *"Lint is very useful to tell in advance what is going wrong"* (P1). The participants stated that the linter is very efficient in detecting code-level performance issues, *"Lint is very good at finding patterned issues that follow a specific rule and can be solved very simply"* (P2). When asked why they want to anticipate performance problems, the participants reported that performance issues should not be noticed by users, "*it is always better to detect eventual problems before they are reported by end-users or managers.*"(P10). In fact, when end-users notice performance issues, they can uninstall the app or give bad reviews, and from there it can be hard to gain back users confidence. Moreover, the participants explained that once bottlenecks are reported by users, it can be complex to identify their root cause and fix them. *"by using Lint, we try to detect performance issues as soon as possible, because when they occur, they are generally very diffuse so it is hard to detect them with the profiler"* (P5). For instance, *"when I have 8 threads with cross-thread method calls, locating a problem with the profiler becomes very difficult"* (P5).

In that respect, anticipating performance bottlenecks saves also time for developers.

Discussion. Previous studies have shown that developers prefer to manage performance reactively, and hence to wait the problems to occur before handling them, with the objective to gain time [27]. Here, our participants express a different point of view. They explain that when performance bottlenecks occur, they require so much time and effort to be fixed. They are hence in favour of a more pro-actively approach that aims to detect and fix bottlenecks before they occur. It is important to transmit this information to developers community, and especially novice Android developers. The latter may be unaware of the complexity of locating and fixing performance bottlenecks, thus they can make wrong strategic choices. As there should be a trade-off between reactive and proactive approaches, we also encourage future works to make real world comparisons between them.

4.1.3 **Linter is easy to use.** Many developers said that they use the performance checks of Android Lint because they found it simple and easy to use (P2, P4, P6, P8, P11, P14). Indeed, Android Lint is already integrated in the IDE and all the checks, including performance, are enabled by default: *"it is built into Android studio, the checks performed by the linter appear automatically so we get those benefits just kind of immediately through the syntax highlighting"* (P2). Hence, the usage of the linter is seamless and effortless: *"it is just built into the tooling so well and so seamlessly, and it does not really cost me anything to have Lint running"* (P2). The participants also appreciated the integration in other development tools: *"we can easily integrate it in the Gradle and continuous integration, so we can ensure that performance and other Lint checks are respected"* (P6).

Discussion. This benefit is more related to the use of the linter itself rather than to performance concerns. Previous studies showed that static analysers should be easy and seamless to encourage developers to use them [25]. Our findings confirm this as participants stated clearly that they use the linter because it does not cost them anything. Now, Android Lint has an additional privilege by being integrated in the official IDE and activated by default and this motivated developers to adopt it. This fact aligns with previous works [17, 25, 36] where researchers suggested that the integration in the development workflow helps static analysers to accumulate the trust of developers.

4.1.4 **Linter can develop the performance culture.** In development teams, developers can have different perceptions and levels of interest regarding performance. Thus, it sounds important to rely on a linter for enforcing performance good practices (P1, P5, P7, P10). The use of a linter ensures that all team members follow the same performance guidelines: *"we have to make sure that everyone respects the rules and understands why we are using the performance checks of Lint"* (P10). On top of that, the performance checks that will occur will certainly be the source of discussions among team members: *" the objective is to share and level our knowledge on performance. When Lint reports performance problems, we can disagree with each other on whether to consider it or not, so we will discuss and understand the problem, then make a wise decision about it"* (P5). That being said, the usage of the linter at a team level is fruitful in many ways. On one side, it allows to keep all the team members at the same page about performance choices. Besides, it arises discussions about performance, and thus enriches the performance culture in the team.

Discussion. Previous study showed that developers use the linter to have an objective tool that avoids endless discussions about code style [38]. The statements of our participants show that the linter itself can trigger discussions in the context of performance. Unlike code style, performance is an important aspect that requires a deep thinking from developers especially in the context of mobile apps. Hence it is normal that developers appreciate the discussions triggered by the linter about it.

4.1.5 **Linter can save the developer's time.** Some participants explained that they use the linter to automate time-consuming tasks (P2, P3, P9). In particular, the linter can fill in for repetitive tasks: *"Lint helps you to save time in a lot of ways. One that comes to my mind is the identification of unused resources. The linter saves me a lot of time as I don't have to cross-check all resources manually"* (P2). Indeed, keeping unused resources is a performance bad practice that increases the APK size. Developers used to manually check and remove all the useless resources, which can be tedious and error prone when there are many resources and when the app is built with many flavors. Using Android Lint to detect unused resources then helps developers and saves their time. Another task where the usage of linter helps saving time is code review: *"it is a quick and easy code review. It gives you a bunch of information. Now, whether I want to implement those messages or not, that is another decision"* (P3). Code review is an important repetitive task in software development that can be very time consuming, especially at team level. As stated by the participants, the linter can partially automate this task, so that developers do not have to review trivial performance issues and can therefore focus on important aspects.

Discussion. Linters are also known for saving time in contexts where eventual issues can be automatically detected, *e.g.,* bug detection [25]. The particularity of the cases reported by our participants is that on top of saving time by detecting eventual issues, the linter automates repetitive tasks. As these tasks are concrete and concern all types of Android apps, this can be more appealing for developers to adopt the linter.

4.1.6 **Linter contributes to increase credibility and reputation.** Developers always want to maintain a good reputation among their peers, and the linter can help them to do that (P1, P6, P7). First, developers need to keep their credibility among colleagues: *"I have to use Lint before making a release to make sure that my work meets the expectations of my superiors and colleagues"* (P6). Another context where reputation is important is open-source projects: *"before releasing code to open-source, I am required to check that all the warnings are away and that my code has a high quality"* (P1). (P7) added: *"I work on a company where we do some open-source projects. Before publishing code I always have to run the linter to make sure of not making obvious performance mistakes. This impacts not only my credibility but also the image of the company"* (P7).

Discussion. Since performance is critical in Android apps, making obvious mistakes can seriously affect the credibility of the developer among her peers. Hence, the linter performance rules can give a valuable support for Android developers to maintain their reputation.

4.1.7 **Linter raises the awareness of the app performance.** Android apps are nowadays very complex software systems, and it is hard for developers to be aware of the implications of their development choices in terms of performance. Some participants stated that they use the linter to always be aware of their app performance (P3, P5). *"Sometimes Lint will catch performance aspects that I did not really think about, and so it will give me a time or a moment to think about and decide"* (P3). This shows that the linter messages incite developers to carefully think and give more attention to performance. This awareness can be particularly important when developers are working alone on the project, *"if I am the only developer, for me this thinking is critical and Lint is a must-have"* (P3). This applies also to the cases where the issues reported by the linter are not applicable: "*in some cases I am not able to apply the changes requested by Lint. But still I need to have a good and valid reason for this. So I am aware of the trade-off I made*" (P5).

Discussion. It is important to distinguish the awareness of app performance with the learning of performance good practices. Here the linter incites the developers to think and understand their app performance. When developers understand their apps, they can make decisions or solve eventual problems more easily.

4.1.8 **Linter eases performance debugging.** Some participants did not only use the linter to directly improve performance, but they also obeyed non-performance rules to ensure high code quality, and consequently ease eventual performance profiling and debugging (P9). *"By using Lint and other static analysers like Sonar, I ensure that my code is well designed and readable. So when a performance issue is reported, debugging and profiling becomes easier. I also can easily and quickly apply the fixes"* (P9). The linter then also helps to build clean and maintainable apps that further developers can easily debug and refactor for improving performance.

Discussion. This represents a benefit of using the linter in general, and not related to the usage of performance checks. However, it is still interesting to observe that some developers have a deep understanding of the software development process. Linters cannot prevent all performance bottlenecks, bugs or other issues. Therefore, developers should always keep their code clean and maintainable, because it makes further maintenance operations easier.

4.2 How do Android developers use linters for performance purposes?

4.2.1 **Linter integrates along the project life cycle.** The participants reported different strategies to use the linter through the project life cycle in order to keep their apps effective. In the remainder, we report on the strategies they identified.

Integrating from scratch. Many participants reported that they prefer using the linter from the project startup (P2, P5, P9, P10, P11, P13). Participant P5 explained that, when starting a project from scratch, she tries to keep the code very clean by considering all the Lint checks. When asked about the configuration adopted in this case, the participants said that they keep the default configuration provided by the linter, *"in this situation I do not need any additional setting, Lint is configured by default"* (P9). We also asked these participants about the motivation behind this strategy. They explained that, when the project advances without the linter, it is more difficult to control performance *a posteriori*. For instance: *"we had a case where we retrieved a project that was built by another team without Lint. We have got thousands of errors and warnings. So we were less motivated to put back Lint and recover the app performance"* (P7). Indeed, it is easy with this strategy to motivate developers to respect performance checks because the code base is clean and there is no existing debt to tackle.

Targeting milestones. Five participants mentioned that they extensively use the linter at the end of features or for releases (P1, P5, P6, P7). *"I never use Lint in the beginning of the project or while prototyping. I use it for releases to make sure that my code meets the expectations"* (P6). As for features: *"towards the end of adding a new feature, I will run through Lint then I will go through all of them and I determine whether or not I want to spend the time to do it"* (P3). When asked about the configuration used for this strategy, participant P5 stated: *"we have different Lint profiles, in the release profile we activate additional rules"*. This strategy allows developers to go fast while producing new features or prototyping without hindering the final app performance.

Adopting performance sprints. Two participants reported that they dedicate sprints for improving the app performance (P5, P12). Participant P12 stated: *"while working, we do not have concrete performance objectives. But when we notice that the app starts lagging, we decide to dedicate time for this and improve the app performance"*. As for participant P5: *"generally while coding, we try to respect the performance checks just as other Lint checks. Then, we regularly program performance sprints and there we will be specifically interested in performance rules"*. While the strategy reported by participant P12 is purely reactive, the strategy of participant P5 is still proactive.

Improving performance incrementally. One participant explained how she deals with legacy code where the linter was not used before (P9). *"I configure Lint to incrementally increase the app performance. I define a baseline, then I work to decrease this baseline progressively. I also try to ensure that the new code is always more effective than the old one"* (P9). Android Lint allows to define a baseline file—*i.e.*, a snapshot of the current warnings of the project. The baseline allows to use Lint without addressing old issues. When asked about how the incremental improvement happens, the participant P9 replied *"I change the severity of some checks, for example I configure some rules to block the compilation"*.

Discussion. Integrating the linter from project start-up is commonly advised [38]. Moreover, previous studies show that developers are less likely to fix the linter warnings on legacy code [14]. The statements of some participants are aligned with this common wisdom. However, the strategies (b) and (c) show that developers can adopt the linter differently according to their work style. In particular, developers who are prototyping or are in rush to release can adopt strategy (b) and apply the linter after finishing their core development. Interestingly, developers who prefer to manage

performance reactively can also leverage the linter by following strategy (c). Finally, strategy (d) shows that the configurability of the linter maximises its chances to be adopted.

4.2.2 **Linter can be used under different settings.** Some participants reported using the linter individually, while others explained how they use it with their team.

Supporting individual development. Half of the participants reported that the usage of the linter was a personal choice (P1, P3, P5, P8, P9, P11, P13). *"I only run Lint as myself as part of a review that I want to do"* (P3). These participants usually use it from the IDE interactively: *"It is through Android studio interactively"* (P13).

Checking performance rules in a team. Other participants reported that the usage of the linter for performance purposes was required on a team level (P2, P4, P6, P7, P9, P10, P12, P14). *"In the team, Lint avoids accumulating problems. We have a defined set of rules and every team member must respect them"* (P10). In this cases, the linter is generally a step in the continuous integration chain: *"it is set up with continuous integration, so Lint runs after every commit and you will get a report. Then, you can choose to look at it or not"* (P2).

Discussion. The reported settings of using the linter do not apply exclusively for performance. The participants explanations underline the importance of the linter interactivity and integration in the development workflow.

4.2.3 **Linter prioritises performance aspects.** Many participants said that while they use the linter, they prioritise performance related issues (P2, P3, P5, P6, P9, P13). For instance: *"there are so many different checks but I would say performance usually catches my eye"* (P2). Some participants gave also distinct priorities to different performance aspects: *"if it is anything about threading, I will take a look at it and review it before deciding if I want to fix it or not"* (P3), *"I give so much importance to UI performance and all memory-related issues"* (P13). Some participants expressed these priorities with a configuration: *"I changed the severity of rules that interest me, so they block the compilation"* (P9).

Discussion. Each app can have different specificities and needs in terms of performance. Thanks to configurability, the linter can help developers to focus on performance aspects that sound relevant and critical for them.

4.3 What are the constraints of using linters for performance purposes?

The constraints reported by participants were structured around two main topics: (i) social challenges and (ii) tool limitations.

4.3.1 **Linter faces social challenges.** The participants reported cultural elements that make the use of linter for performance challenging. The participants encountered these issues in their work environment, with colleagues or superiors.

Static analysis is not suitable for performance. Many participants described that developers generally think that static analysis is not suitable for improving performance (P1, P2, P3, P4, P7, P12). Participant P1 stated: *"I think that there is a gap in understanding why static analysis is useful for performance"*. Participants explained that this mindset is due to the nature of static analysis: *"because Lint is only looking at the code. Some developers feel that there should be a better tool that analyses the app while running"* (P3). Other participants thought this gap is due to the complexity of performance issues: *"for the actual real-world bottlenecks that most apps face, it is not the Lint that will help you. Performance issues are very complicated or have multiple causes that cannot be pinpointed to a one line of Java code"* (P2). Participant P4 stated that this gap may be due to the confusion of the term "performance issue": *"For each performance issue there is a root cause and an observation. The term performance issue is often used to refer to both. But it is necessary to distinguish them. The default Lint rules contain some basic and trivial root causes, which could statically be identified. But in some cases you have an observation and you cannot guess the root cause. So here Lint cannot help you. To sump up, Lint requires you to define in advance what you are looking for, it is hard to use it to match the observation and the cause"*.

Nobody complained. Many participants reported that they regularly deal with colleagues and superiors who believe that performance should be managed reactively (P1, P3, P4, P5, P8, P9, P12). For example, participant P5 stated that the common rule in her environment is *"only when the superiors or the end-users complain that the app is lagging or not smooth, we say ok we have to care about performance"*. Participant P5 highlighted a case where this pressure came from a superior: *"performance refactoring is a back-office task that the product owner cannot see. It is hard to negotiate these tasks"*. Some participants underlined that this mindset is particularly tied to performance more than any other software quality aspect: *"with performance you do not want to do a lot, you want to make sure that you are really looking at the issue and not trying to over optimise the code"* (P3).

We do not have time. This mindset is very related to the previous one. However, we observed cases where the developers explain that the performance checks of the linter are not considered only for time constraints without any explicit agreement on the management of performance reactively (P1, P6, P7). Participant P1 reports observing this mindset in many companies: *"why waste time on performance rules? Let us move ahead and we will figure about this later. Unfortunately that later nevere happens, or comes only when the problem is big enough"*.

Performance is not important in our case. Some participants reported working in contexts where performance was considered irrelevant (P1, P8, P12). Participant P1 reports experiencing this situation in young projects: *"when you build a small app and do not know whether it will scale or not, it does not seem useful to spend time in making sure that static analysis of performance is done right"*. Participant P8 described a case where performance was considered irrelevant for a type of apps: *"we did very basic app development and not particularly hardware-based development. We developed Uber clones, and all those apps did not require any specific performance from the device"*.

The linter rules are irrelevant. Two participants described that some performance checks are considered irrelevant (P2, P12). Participant P12 gave examples of Lint performance rules that do not

really have an impact: *"OverDraw is a rule that used to be relevant a long time ago, but now with powerful smartphones it is not anymore. Moreover, the Android system detects all these trivial issues and fixes them automatically. Developers are obsessed with Overdraw, it became a cult actually, but this is not what really hinders the performance of your app"*. The problem OverDraw occurs when a background drawable is set on a root view. This is problematic because the theme background will be painted in vain as the drawable will completely cover it.

Discussion. Calcagno *et al.* [15] have already referred to the social challenge while describing their experience in integrating a static analyser into the software development cycle at Facebook. Some of the reported challenges, *e.g.,* we do not have time, apply to linters in general. However, the other mindsets are particularly resistant to the use of linters for performance. The belief that static analysis is not suitable for performance seems to be prevalent, six participants mentioned it explicitly. Developers are used to linters as tools that report minor issues like styling violations and deprecations. They are not aware enough of the capabilities of static analysis in detecting performance issues. Tool makers should put more efforts on communicating about linters as tools that can accompany developers in different development aspects. The mindset that performance should be managed reactively confirms previous observations about Android apps development [27]. This finding shows that Android developers still lack understanding about the implications of performance bottlenecks. As for linter rules that are considered irrelevant, this incites the research community to dig deeper into the impact of these practices. For many bad practices like OverDraw, we still lack precise measurements of their penalties on performance in real world contexts.

4.3.2 **Linter suffers from limitations.** The participants reported several linter limitations that make it complicated to use it for performance purposes.

Not well presented. Several participants mentioned that the linter rules are not well organised or explained (P1, P4, P6, P11, P13, P14). Interestingly three participants said explicitly that, for a long time, they did not even know that Android Lint had performance related checks (P1, P11, P14), *"I did not know there are different categories in Lint. For me it is just Lint, I do not distinguish between them"* (P1). Furthermore, participant P14 explained that in the beginning she did not know that some rules are related to performance, and thus she treated them as other checks like typography. Other participants complained about the unclarity of the messages, *"it is not always clear, for example performance checks about graphics, I cannot really understand them at all"*. On top of that, some participants found that rules are not well organised: *"there is a hierarchy with levels of importance, but I find it useless, it does not help me. So if I want to focus only on performance aspects, I have to search everywhere"* (P6). The same participant underlined the unclarity of the priorities given by the linter to the checks: *"I try to obey, but I do not really understand the logic behind the priority and severity"*. Indeed, Android Lint does not give explanations about the priority and severity attributed to each check, so we cannot understand the rationales behind them.

Imprecision. Some participants complained about the imprecision of the detection performed by the linter for performance checks (P9, P10, P13). Participant P9 described situations where the code contained a performance bad practice, but the corresponding linter check was unable to detect it: *"in some drawing methods, I was calling a method that called another one that made an intensive computing with memory allocations. Lint did not detect this as DrawAllocation, it actually does not look so deep into the code"*. Other participants reported false positives in performance checks. Participant P13 said: *"I regularly have false warnings about unused resources when the resource is used via a library"*, and participant P10 stated: *"Lint indicates an unused resource but the image is actually used with a variable"*.

Poverty. Some participants mentioned that the linter is not very rich with performance checks (P5, P6, P9). Participant P5 stated: *"I do not see so many performance-related Lint checks. And the existing ones are so generic"*. In the same vein, participant P9 said: *"I rarely see suggestions or important warnings about performance aspects. Very few!"*. Furthermore, the participants complained about the absence of linter checks for Kotlin (P6, P9).

The difficulty of writing new rules. Participant P2 described her trial to write a linter check and the difficulties she faced: *"I wanted to write specific Lint rules and after a few trials I ended up discovering that it is difficult to define something general enough to warrant a Lint check. Also, the effort put in to build a custom Lint check is pretty high. That is why it is not a common tactic for development teams especially on a per project basis"*. The participant pointed also the complexity of using the created rule in a team: *"to build a Lint check, then distribute it to the team, then have the whole team use it, is difficult"*.

Discussion. The fact that at least three participants reported that for a long time they used the linter without noticing that it has performance checks was striking. Tool makers have to work more on showcasing different checks categories. Also, the linter messages should highlight more the impact of performance practices to motivate developers to consider them seriously. The imprecision is a common limitation of linters [25], and performance checks are no different in that respect. Similarly, the participants statements about the difficulty to write linter rules align with previous works [17]. As many other tools, Android Lint provides the possibility to write new rules but this task is complex and time-consuming. Thus, developers are not motivated to write their own rules.

5 IMPLICATIONS

We summarise in this section the several implications of our results for developers, researchers, and tools creators. Our findings are based on the usage of linters for performance purposes in Android. Nevertheless, they can also apply to other development platforms.

5.1 For Developers

Our results provide motivations for developers to use linters for performance and show them how to maximise their benefits.

Benefits. Developers can find several benefits in using the linter for performance. In particular, developers can use the linter with performance checks to:

• Learn about the mobile framework and its performance constraints,
• Anticipate performance bottlenecks that can be arduous to identify and fix,
• Develop the performance culture in the team,
• Save time by automating concrete repetitive tasks,
• Save their reputation among peers,
• Increase their awareness and understanding of their apps performance.
Developers should also be aware that the usage of the linter is seamless and can be integrated in along the development workflow.

Usage fashions. Our participants recommend to use the linter for performance in the following ways:
• From project startup to motivate developers to keep the code clean,
• Only before releases to dedicate the early development stages only for prototyping and making the important features,
• In performance sprints: developers can configure the linter to focus only on performance in some sprints. This approach works also for developers who prefer to manage the performance reactively,
• Improve performance incrementally: developers should configure the linter carefully on legacy code to avoid chaos and developers discouragement,
• Individually in an interactive way in the IDE or in a team with the continuous integration,
• Prioritising performance aspects: developers can configure the linter to focus on performance aspect that interest them and fit with their app needs.

5.2 For Researchers

Our findings confirm hypotheses from several previous works and open up perspectives for new research directions:
• We confirm that the mindset of managing performance reactively is prevalent [27]. As there should be a trade-off between reactive and proactive approaches, we encourage future works to make real world comparisons between them;
• We confirm that some developers are unaware of the importance and impact of performance bad practices in mobile apps [21]. Researchers should invest in popular scientific works about this topic in order to transmit the message to developers;
• Some developers challenge the relevance and impact of performance bad practices. We therefore encourage future works to investigate and provide precise evidences about the impact of such practices.
• Some developers are eager to consider more performance-related checks. This should incite researchers to identify and characterise more practices that can hinder the performance of mobile apps.

5.3 For Tool Creators

Our findings confirm the importance of some linter features and highlight new needs and challenges:
• Our findings align with previous works that suggest that simplicity and integration in the development workflow help static analysers to increase the trust of developers [17, 25, 36]. We encourage tool creators to ease the integration of their linters in development tools;
• Our findings show that linters should be more clear and explicit about the categories of checks. Clarity is also required in the explanations of the potential impacts of performance checks. We cannot expect from developers to seriously consider the rules if we do not provide enough information about their nature and impacts;
• Providing the possibility to write linter rules is not enough. Writing a linter rule should be simple and less time-consuming to motivate developers to do it;
• Tool makers should put more efforts in communicating about the capabilities of static analysis in detecting performance bad practices;
• Given the benefits reported by participants, we invite more tool makers to include performance-related checks in their linters.

6 THREATS TO VALIDITY

We discuss in this section the main issues that may threaten the validity of our study.

Transferability. : One limitation to the generalisability of our study is the sample size. The sample size is not large and thus it may not be representative of all Android developers. To alleviate this fact, we interviewed highly-experienced Android developers. Nonetheless, this selection may also introduce a new bias to the study. As a matter of fact, the benefits and constraints reported by junior Android developers can be different. We would have liked more participants. However, transcribing interviews is a manual and time-consuming task. Hence, having more interviews may involve more workload and would affect the accuracy and quality of the analysis. We found our results valuable and enough for theoretical saturations. The study conducted by Tómasdóttir *et al.* [38], which we discuss in our related works, approaches a similar topic with a similar sample size (15 participants). Another possible threat is that we only interviewed Android developers who use the linter for performance purposes. Android developers who use the linter without performance checks may also have their word to say about the limitations of using a linter for performance. Thus, more studies should be conducted to understand why some Android developers use the linter and disable performance checks. As we focused our study on Android Lint, our results cannot be generalisable to other Android linters. However, we believe that the choice of Android Lint was sound for two reasons. First, it is a built-in tool of Android Studio and is activated by default, thus a large proportion of Android developers should be using it. This fact was confirmed in our preliminary online survey, where 97 % of the participants who used the linters were actually relying on Android Lint [12]. Secondly, it is the only linter that has performance checks specific to the Android platform, and this detail is the core of our study.

Credibility. : One possible threat to our study results could be the credibility of participants answers. We interviewed developers who have a strong knowledge about the topic. However, we cannot be sure that their answers were based on real experience or knowledge acquired from external resources. To alleviate this issue, we tried always to ask for details and relate to developers project and

working environment. Also, we emphasised before the interviews that the process is not judgmental.

Confirmability. : One possible threat could be the accuracy of the interviews analysis and particularly the coding step. We use a consensual coding to alleviate this threat and strengthen the reliability of our conclusions. Each interview has been encoded by at least two authors. Initially, every author coded the interview independently to avoid influencing the analysis of other authors. Afterwards, the authors discussed and compared their classifications.

7 RELATED WORK

Mobile performance bad practices. Mobile performance bad practices —*a.k.a.* code smells—have been addressed in various studies. Many studies investigated the energy aspect, they considered code smells related to networking, sensors, non-sleep, and general [28, 33, 34, 39–41]. While all these studies relied on app repositories and forums, Linarez *et al.* [26] used power profiling and API calls analysis to identify energy code smells. Other studies focused on identifying and characterising mobile performance smells and detecting them. Guo *et al.* [20] studied resource leaks in Android apps. They proposed the Relda tool to detect resources that are exclusive, memory-consuming, or energy consuming. Liu *et al.* [29] studied 70 performance bugs from eight Android apps and identified their characteristics and common patterns. They also implemented PerfChecker, a static analyser that detects the identified performance bugs. The main identified bugs are lengthy operations in the main thread, wasted operations for GUI, and frequently-called heavy callbacks. For bug characteristics, they found that performance bugs are more difficult to debug and fix than non-performance bugs. In terms of debugging, profilers and performance measurement tools have demonstrated to be more helpful than traditional stack trace information. Additionally, works that proposed catalogs of mobile-specific code smells also included performance related issues [21, 35]. As mobile performance smells were identified, researchers were interested in assessing their real impact on different performance aspects. Hecht *et al.* [22] conducted an empirical study about the individual and combined impacts of three Android performance smells. They measured the performance of two apps with and without smells using the following metrics: frame time, number of delayed frames, memory usage, and number of garbage collection calls. The measurements showed that refactoring the Member Ignoring Method smell improves the frames metrics by 12.4%. Carette *et al.* [16] studied the same code smells but focused on the energy impact. They analysed 5 open-source Android apps and observed that in one of them the refactoring of the three code smells reduced the global energy consumption by 4, 83%.

Performance management. Linarez *et al.* [27] surveyed developers to identify the common practices and tools for detecting and fixing performance issues in Android open-source apps. Based on 485 answers, they deducted that most of developers rely on reviews and manual testing for detecting performance bottlenecks. When asked about tools, developers reported using profilers and framework tools, only five of them mentioned using static analysers. Developers were also openly questioned about the targets of their performance improvement practices. From 72 answers, the study established the following categories: GUI lagging, memory bloats, energy leaks, general performance, and unclear benefits. With the aim of helping developers understand and predict performance problems in mobile apps. Nistor *et al.* [30] proposed SunCat, a tooled-approach that based on a run with small inputs explains how would an app behave with large inputs. In five apps, SunCat identified 29 usage scenarios and five confirmed performance problems.

Qualitative studies on linters. Tómasdóttir *et al.* [38] conducted a qualitative study to investigate the benefits of using linters in a dynamic programming language. They interviewed 15 developers to understand why and how JavaScript developers use ESLint [7] in OSS. They found that linters can be used to augment test suites, they spare newcomers' feelings when making their first contribution, and save time that goes into discussing code styles. Christakis and Bird [17] conducted an empirical study combining interviews and surveys to investigate the needs of developers from static analysis. Among other results, they found that performance issues are the second most severe code issues that require an immediate intervention of developers. Performance issues were also in the top four needs of developers. Johnson *et al.* [25] conducted 20 interviews to understand why developers do not use static analysis tools like FindBugs [8] to find bugs. They found that all the participants are convinced by the benefits of static analysis tools. However, the false positives and warnings presentation are the main barriers of tools adoption.

8 CONCLUSION

We investigated in this paper the benefits and the constraints of using linters for performance purposes in Android apps. We conducted a qualitative study based on interviews with experienced Android developers. Our results provide motivations for developers to use linters for performance and share with them how to make this usage the most beneficial. Our findings highlight also the current challenges of using linters for performance. These challenges open up new research perspectives and show new needs for tool makers.

ACKNOWLEDGMENTS

The authors would like to thank the interviewees for their time and involvement. They also thank all developers who participated to the online survey that motivated this study.

REFERENCES

[1] 2015. Mobile Stats. https://www.soasta.com/blog/22-mobile-web-performance-stats/. [Online; accessed April-2018].
[2] 2016. Mobile App Retention Challenge. https://dazeinfo.com/2016/05/19/mobile-app-retention-churn-rate-smartphone-users/. [Online; accessed April-2018].
[3] 2018. Android Dev. https://www.reddit.com/r/androiddev/. [Online; accessed April-2018].
[4] 2018. Android Lint. https://developer.android.com/studio/write/lint.html. [Online; accessed March-2018].
[5] 2018. CheckStyle. http://checkstyle.sourceforge.net/. [Online; accessed April-2018].
[6] 2018. Detekt. https://github.com/arturbosch/detekt. [Online; accessed April-2018].
[7] 2018. ESLint. https://eslint.org/. [Online; accessed April-2018].
[8] 2018. FindBugs. http://findbugs.sourceforge.net/. [Online; accessed April-2018].
[9] 2018. Infer. http://fbinfer.com/. [Online; accessed April-2018].

[10] 2018. Ktlint. https://github.com/shyiko/ktlint. [Online; accessed April-2018].
[11] 2018. PMD. https://pmd.github.io/. [Online; accessed April-2018].
[12] 2018. Technical Report. https://zenodo.org/record/1320453.
[13] Steve Adolph, Wendy Hall, and Philippe Kruchten. 2011. Using grounded theory to study the experience of software development. *Empirical Software Engineering* 16, 4 (2011), 487–513.
[14] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008).
[15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.
[16] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2017. Investigating the energy impact of android smells. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 115–126.
[17] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 332–343.
[18] John W Creswell and J David Creswell. 2017. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.
[19] Barney G Glaser and Judith Holton. 2007. Remodeling grounded theory. *Historical Social Research/Historische Sozialforschung. Supplement* (2007), 47–68.
[20] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 389–398.
[21] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. 2017. Code Smells in iOS Apps: How do they compare to Android?. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 110–121.
[22] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, 59–69.
[23] Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the Software Quality of Android Applications along their Evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 12.
[24] Siw Elisabeth Hove and Bente Anda. 2005. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software metrics, 2005. 11th ieee international symposium*. IEEE, 10–pp.
[25] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 672–681.
[26] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2–11.
[27] Mario Linares-Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in android apps. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 352–361.
[28] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2013. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*. IEEE, 2–10.
[29] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1013–1024.
[30] Adrian Nistor and Lenin Ravindranath. 2014. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 282–292.
[31] Daniel G Oliver, Julianne M Serovich, and Tina L Mason. 2005. Constraints and opportunities with interview transcription: Towards reflection in qualitative research. *Social forces* 84, 2 (2005), 1273–1289.
[32] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Lightweight detection of Android-specific code smells: The aDoctor project. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 487–491.
[33] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 2011. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 5.
[34] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. 2012. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 267–280.
[35] Jan Reimann, Martin Brylski, and Uwe Aßmann. 2014. A Tool-Supported Quality Smell Catalogue For Android Developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung – MMSM 2014*.
[36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 1 (2015), 598–608.
[37] Christiane Schmidt. 2004. The analysis of semi-structured interviews. *A companion to qualitative research* (2004), 253–258.
[38] Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie van Deursen. 2017. Why and how JavaScript developers use linters. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 578–589.
[39] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. 2012. Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs.. In *HotPower*.
[40] Jack Zhang, Ayemi Musa, and Wei Le. 2013. A comparison of energy bugs for smartphone platforms. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*. IEEE, 25–30.
[41] Lide Zhang, Mark S Gordon, Robert P Dick, Z Morley Mao, Peter Dinda, and Lei Yang. 2012. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 363–372.